Appendix F: Programming Conventions

The LogiQL language and compiler give a great deal of latitude to programmers to express their ideas. In particular, the choice of names, capitalization, and use of whitespace and comments are relatively unconstrained. Nevertheless, there are some advantages to using uniform conventions: Programs become more readable, errors are more easily detected, and training is facilitated. This appendix describes the set of conventions used in this book. First, a descriptive overview is given, and then summary lists are provided. A short exercise completes the appendix.

OVERVIEW

In this book, we adopted the convention of starting the names of user-defined object types with a capital letter, using a noun phrase to name the type. Typically, these types have been entity types represented by entity predicates (e.g., `Monarch`, `MalePerson`), but we may also use a predicate for a domain-related value type (e.g., `PersonTitle`). In contrast, we started property predicate names with a lowercase letter, using a verb phrase. In both situations, we rendered the remainder of the name in camelCase (e.g., `hasGivenName`), where words after the first are appended, beginning with a capital letter. Remember that LogiQL is case sensitive, so you need to ensure that you use the appropriate case when referencing a predicate. Built-in predicates for types and functions (e.g., `string`, `count`) always start with a lowercase letter.

There are some syntactic rules that must be obeyed. For example, a colon (':') must be used to separate the arguments of a refmode predicate, such as hasGenderCode(g:gc). When declaring functional predicates other than refmode predicates, we must enclose the keyspace arguments in square brackets (e.g., fatherOf[p1]=p2). Predicates that are not functional have their arguments in parentheses, for example, isParentOf(p1,p2).

If a functional predicate is binary, we have often used a *functionOf* style of naming to render a natural reading. For example, the equation genderOf[m]=g (i.e., "the gender of Monarch m is g") reads more naturally than hasGender[m] = g. In some cases, a preposition other than "Of" is more natural (e.g., nrBooksAuthoredBy[p]=n). We have tended to use short names for predicate arguments, often using one or just a few suggestive letters and maybe a digit. Some people like to use more descriptive names for the arguments. For example, instead of isParentOf(p1,p2) they might use isParentOf(parent,child).

In LogiQL, predicates are typed, so each argument of the predicate is constrained in its declaration to belong to a specific, named type. For example, suppose you make the following type declaration for a predicate named runs:

```
runs(p, c) -> Person(p), Company(c).
```

Now suppose that you want to record facts about people running races. If you try to use the same predicate name runs for these facts, you will get an error because the type of the runs predicate would then be ambiguous:

```
runs(p, r) -> Person(p), Race(r). // Error!
```

One way to fix this error is to expand the predicate name to include the type name of the second argument:

```
runsRace(p, r) -> Person(p), Race(r).
```

You could also rename the original runs predicate to runsCompany, but this is not required to resolve the error. Now suppose you also wish to record facts about horses running races. If we add the following declaration, we get another error, because the type of runsRace would be ambiguous:

```
runsRace(h, r) -> Horse(h), Race(r). // Error!
```

One way to fix this error is to expand the predicate name to include the type names of *all* its arguments:

```
horseRunsRace(h, r) -> Horse(h), Race(r).
```

If you wish, you could do this also for the previous predicates, renaming them as `personRunsCompany` and `personRunsRace`, although this is not required.

In the above examples, we embedded the type names inside the predicate name simply by concatenation. Another naming style that is often used is to prepend the type names to the rest of the predicate name, using a colon (':') separator. Applying this naming style to all three of the above predicates leads to the predicate names `person:company:runs`, `person:race:runs`, and `horse:race:runs`. The three predicates would then be declared as follows:

```
Person(p), person:company:runs(p, c) -> Company(c).
Person(p), person:race:runs(p, r) -> Race(r).
Horse(h), horse:race:runs(h, r) -> Race(r).
```

This naming style ensures distinct predicate names and facilitates searching for predicates based on the same type(s). It is also convenient for refmode predicates, since these may now be named by prepending the entity type name to a short refmode name. For example, the following declarations

```
Monarch(m), hasMonarchName(m:mn) -> string(mn).
Country(c), hasCountryCode(c:cc) -> string(cc).
State(s), hasStateCode(s:sc) -> string(sc).
```

may be rephrased as follows:

```
Monarch(m), Monarch:name(m:mn) -> string(mn).
Country(c), Country:code(c:cc) -> string(cc).
State(s), State:code(s:sc) -> string(sc).
```

However, in most cases this naming style makes the code longer and less natural to verbalize. For example, compare the following two declarations with their alternatives below:

```
hasGivenName(p, gn) -> Person(p), string(gn).
fatherOf[p1] = p2 -> Person(p1), Person(p2).
```

```
person:string:hasGivenName(p, gn) -> Person(p),
    string(gn).
person:person:hasFather[p1] = p2 -> Person(p1),
    Person(p2).
```

Note that LogiQL allows the same predicate to be declared on multiple subtypes of a common supertype. For example, in the below program `isLicensed` is constrained to instances of `Doctor` and `Driver`, which are overlapping subtypes of `Person`:

```
Person(p), person:name(p:n) -> string(n).
Driver(d) -> Person(d).
Doctor(d) -> Person(d).
lang:isEntity[`Driver] = true.
lang:isEntity[`Doctor] = true.
isLicensed(d) -> Driver(d).
isLicensed(d) -> Doctor(d).
```

Here, valid arguments to the `isLicensed` predicate are constrained to be people who are both drivers and doctors (i.e., instances of the intersection of `Driver` and `Doctor`). The LogiQL compiler effectively infers `Person` as a common type for this predicate. If instead you intend that there are two different predicates for being licensed, one for driving an automobile and one for practicing medicine, then you must name these predicates differently:

```
isLicensedToDrive(d) -> Driver(d).
isLicensedToPractiseMedicine(d) -> Doctor(d).
```

USE OF INDENTATION AND WHITESPACE

1. Surround occurrences of arrows, either right arrows (‘`->`’) or left arrows (‘`<-`’), with spaces.

2. In executable code, follow commas (‘`,`’) with a space. For stand-alone **predicate signatures**, the spaces may be elided (i.e., `hasGivenName(p,gn)`).

3. When using the functional notation, surround occurrences of equals signs (‘ = ’) with spaces.

4. Surround occurrences of the disjunction operator (‘`;`’) with spaces.

5. Indent the second and subsequent lines in a rule or constraint four spaces to the right of the first line.
6. Keep individual lines to 80 characters or less.
7. When splitting a rule or constraint across lines, put the left or right arrow at the end of the last line of the head.

COMMENTS

1. Comments in this book that are associated with examples indicate how to verbalize the commented code.
2. LogiQL comment text should be separated from the comment indicators by at least one space.

NAMING AND CAPITALIZATION

1. If any identifier comprises multiple words, capitalize all words after the first.
2. Lowercase letters start the names of non-entity predicates, such as property predicates. These take the form of verbs or verb phrases.
3. Entity type names begin with an uppercase letter and take the form of a noun or a noun phrase.
4. Names for refmode predicates should begin with "has" followed by the entity type name followed by an indicator of the refmode's representation, such as "Name" or "Code."
5. Express a binary functional predicate in the `functionOf` style, where the name of a property is followed by "`Of`" or another relevant preposition such as "`By`".
6. Names of non-functional predicates with two or more arguments can take the form of a verb, possibly adjacent to the types of the arguments. That is, a type might follow the verb, or the verb might be surrounded by two type names.
7. Variable names are typically one or two letters, beginning with a letter descriptive of the type or purpose the variable plays in the rule or constraint. If the rule is so complex that more descriptive names are required of the variables, it may be better to split the rule or constraint into pieces.

8. Underscores should not normally be used in names. Exceptions to this rule are allowed for occurrences of anonymous variables and predicates and to indicate the positions of arguments in verbalizing predicate names, as used, for example, in Unit 4.1.

Exercise 1: Explain what is wrong with the following code, and revise it to avoid the problem:

```
Person(p), person:name(p:n) -> string(n).
Gun(g), gun:serialNr(g:n) -> string(n).
fired(p1, p2) -> Person(p1), Person(p2).
fired(p, g) -> Person(p), Gun(g).
```

ANSWERS TO EXERCISES

Answer to Exercise 1:

The `fired` predicate is declared to have different types that have no common supertype. To fix this, at least one of the `fired` predicates needs to be renamed. To avoid confusion, it's best to rename both. Here is one solution that expands the predicate names to distinguish their meanings:

```
Person(p), person:name(p:n) -> string(n).
Gun(g), gun:serialNr(g:n) -> string(n).
firedPerson(p1, p2) -> Person(p1), Person(p2).
firedGun(p, g) -> Person(p), Gun(g).
```

Here is another solution that prepends the type names with a colon to the original predicate names:

```
Person(p), person:name(p:n) -> string(n).
Gun(g), gun:serialNr(g:n) -> string(n).
person:person:fired(p1, p2) -> Person(p1),
    Person(p2).
person:gun:fired(p, g) -> Person(p), Gun(g).
```